Ensuring Data Storage Security in Cloud Computing

Cong Wang, Qian Wang, and Kui Ren
Department of ECE
Illinois Institute of Technology
Email: {cwang, qwang, kren}@ece.iit.edu

Wenjing Lou
Department of ECE
Worcester Polytechnic Institute
Email: wjlou@ece.wpi.edu

***Abstract*—Cloud Computing has been envisioned as the next-generation architecture of IT Enterprise. In contrast to traditional solutions, where the IT services are under proper physical, logical and personnel controls, Cloud Computing moves the application software and databases to the large data centers, where the management of the data and services may not be fully trustworthy. This unique attribute, however, poses many new security challenges which have not been well understood. In this article, we focus on cloud data storage security, which has always been an important aspect of quality of service. To ensure the correctness of users' data in the cloud, we propose an effective and flexible distributed scheme with two salient features, opposing to its predecessors. By utilizing the homomorphic token with distributed verification of erasure-coded data, our scheme achieves the integration of storage correctness insurance and data error localization, i.e., the identification of misbehaving server(s). Unlike most prior works, the new scheme further supports secure and efficient dynamic operations on data blocks, including: data update, delete and append. Extensive security and performance analysis shows that the proposed scheme is highly efficient and resilient against Byzantine failure, malicious data modification attack, and even server colluding attacks.**

I. Introduction

Several trends are opening up the era of Cloud Computing, which is an Internet-based development and use of computer technology. The ever cheaper and more powerful processors, together with the software as a service (SaaS) computing architecture, are transforming data centers into pools of computing service on a huge scale. The increasing network bandwidth and reliable yet flexible network connections make it even possible that users can now subscribe high quality services from data and software that reside solely on remote data centers.

Moving data into the cloud offers great convenience to users since they don't have to care about the complexities of direct hardware management. The pioneer of Cloud Computing vendors, Amazon Simple Storage Service (S3) and Amazon Elastic Compute Cloud (EC2) [1] are both well known examples. While these internet-based online services do provide huge amounts of storage space and customizable computing resources, this computing platform shift, however, is eliminating the responsibility of local machines for data maintenance at the same time. As a result, users are at the mercy of their cloud service providers for the availability and integrity of their data. Recent downtime of Amazon's S3 is such an example [2].

From the perspective of data security, which has always been an important aspect of quality of service, Cloud Computing inevitably poses new challenging security threats for number of reasons. Firstly, traditional cryptographic primitives for the purpose of data security protection can not be directly adopted due to the users' loss control of data under Cloud Computing. Therefore, verification of correct data storage in the cloud must be conducted without explicit knowledge of the whole data. Considering various kinds of data for each user stored in the cloud and the demand of long term continuous assurance of their data safety, the problem of verifying correctness of data storage in the cloud becomes even more challenging. Secondly, Cloud Computing is not just a third party data warehouse. The data stored in the cloud may be frequently updated by the users, including insertion, deletion, modification, appending, reordering, etc. To ensure storage correctness under dynamic data update is hence of paramount importance. However, this dynamic feature also makes traditional integrity insurance techniques futile and entails new solutions. Last but not the least, the deployment of Cloud Computing is powered by data centers running in a simultaneous, cooperated and distributed manner. Individual user's data is redundantly stored in multiple physical locations to further reduce the data integrity threats. Therefore, distributed protocols for storage correctness assurance will be of most importance in achieving a robust and secure cloud data storage system in the real world. However, such important area remains to be fully explored in the literature.

Recently, the importance of ensuring the remote data integrity has been highlighted by the following research works [3]–[7]. These techniques, while can be useful to ensure the storage correctness without having users possessing data, can not address all the security threats in cloud data storage, since they are all focusing on single server scenario and most of them do not consider dynamic data operations. As an complementary approach, researchers have also proposed distributed protocols [8]–[10] for ensuring storage correctness across multiple servers or peers. Again, none of these distributed schemes is aware of dynamic data operations. As a result, their applicability in cloud data storage can be drastically limited.

In this paper, we propose an effective and flexible distributed scheme with explicit dynamic data support to ensure the correctness of users' data in the cloud. We rely on erasure-correcting code in the file distribution preparation to provide redundancies and guarantee the data dependability. This construction drastically reduces the communication and storage overhead as compared to the traditional replication-based file

المنارة للاستشارات
www.manaraa.com

distribution techniques. By utilizing the homomorphic token with distributed verification of erasure-coded data, our scheme achieves the storage correctness insurance as well as data error localization: whenever data corruption has been detected during the storage correctness verification, our scheme can almost guarantee the simultaneous localization of data errors, i.e., the identification of the misbehaving server(s).

Our work is among the first few ones in this field to consider distributed data storage in Cloud Computing. Our contribution can be summarized as the following three aspects:

1) Compared to many of its predecessors, which only provide binary results about the storage state across the distributed servers, the challenge-response protocol in our work further provides the localization of data error.

2) Unlike most prior works for ensuring remote data integrity, the new scheme supports secure and efficient dynamic operations on data blocks, including: update, delete and append.

3) Extensive security and performance analysis shows that the proposed scheme is highly efficient and resilient against Byzantine failure, malicious data modification attack, and even server colluding attacks.

The rest of the paper is organized as follows. Section II introduces the system model, adversary model, our design goal and notations. Then we provide the detailed description of our scheme in Section III and IV. Section V gives the security analysis and performance evaluations, followed by Section VI which overviews the related work. Finally, Section VII gives the concluding remark of the whole paper.

II. Problem Statement

A. System Model

A representative network architecture for cloud data storage is illustrated in Figure 1. Three different network entities can be identified as follows:

- User: users, who have data to be stored in the cloud and rely on the cloud for data computation, consist of both individual consumers and organizations.
- Cloud Service Provider (CSP): a CSP, who has significant resources and expertise in building and managing distributed cloud storage servers, owns and operates live Cloud Computing systems.
- Third Party Auditor (TPA): an optional TPA, who has expertise and capabilities that users may not have, is trusted to assess and expose risk of cloud storage services on behalf of the users upon request.

In cloud data storage, a user stores his data through a CSP into a set of cloud servers, which are running in a simultaneous, cooperated and distributed manner. Data redundancy can be employed with technique of erasure-correcting code to further tolerate faults or server crash as user's data grows in size and importance. Thereafter, for application purposes, the user interacts with the cloud servers via CSP to access or retrieve his data. In some cases, the user may need to perform block level operations on his data. The most general forms of these operations we are considering are block update, delete, insert and append.

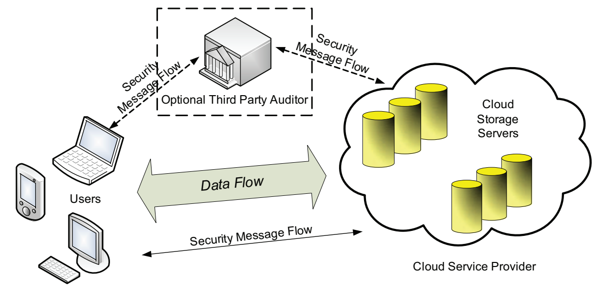

Fig. 1: Cloud data storage architecture

As users no longer possess their data locally, it is of critical importance to assure users that their data are being correctly stored and maintained. That is, users should be equipped with security means so that they can make continuous correctness assurance of their stored data even without the existence of local copies. In case that users do not necessarily have the time, feasibility or resources to monitor their data, they can delegate the tasks to an optional trusted TPA of their respective choices. In our model, we assume that the point-to-point communication channels between each cloud server and the user is authenticated and reliable, which can be achieved in practice with little overhead. Note that we don't address the issue of data privacy in this paper, as in Cloud Computing, data privacy is orthogonal to the problem we study here.

B. Adversary Model

Security threats faced by cloud data storage can come from two different sources. On the one hand, a CSP can be self-interested, untrusted and possibly malicious. Not only does it desire to move data that has not been or is rarely accessed to a lower tier of storage than agreed for monetary reasons, but it may also attempt to hide a data loss incident due to management errors, Byzantine failures and so on. On the other hand, there may also exist an economically-motivated adversary, who has the capability to compromise a number of cloud data storage servers in different time intervals and subsequently is able to modify or delete users' data while remaining undetected by CSPs for a certain period. Specifically, we consider two types of adversary with different levels of capability in this paper:

Weak Adversary: The adversary is interested in corrupting the user's data files stored on individual servers. Once a server is comprised, an adversary can pollute the original data files by modifying or introducing its own fraudulent data to prevent the original data from being retrieved by the user.

Strong Adversary: This is the worst case scenario, in which we assume that the adversary can compromise all the storage servers so that he can intentionally modify the data files as long as they are internally consistent. In fact, this is equivalent to the case where all servers are colluding together to hide a data loss or corruption incident.

المنارة للاستشارات
www.manaraa.com

C. Design Goals

To ensure the security and dependability for cloud data storage under the aforementioned adversary model, we aim to design efficient mechanisms for dynamic data verification and operation and achieve the following goals: (1) Storage correctness: to ensure users that their data are indeed stored appropriately and kept intact all the time in the cloud. (2) Fast localization of data error: to effectively locate the malfunctioning server when data corruption has been detected. (3) Dynamic data support: to maintain the same level of storage correctness assurance even if users modify, delete or append their data files in the cloud. (4) Dependability: to enhance data availability against Byzantine failures, malicious data modification and server colluding attacks, i.e. minimizing the effect brought by data errors or server failures. (5) Lightweight: to enable users to perform storage correctness checks with minimum overhead.

D. Notation and Preliminaries

- $\mathbf{F}$ – the data file to be stored. We assume that $\mathbf{F}$ can be denoted as a matrix of m equal-sized data vectors, each consisting of l blocks. Data blocks are all well represented as elements in Galois Field $GF(2^p)$ for $p = 8$ or 16.
- $\mathbf{A}$ – The dispersal matrix used for Reed-Solomon coding.
- $\mathbf{G}$ – The encoded file matrix, which includes a set of $n = m + k$ vectors, each consisting of l blocks.
- $f_{key}(\cdot)$ – pseudorandom function (PRF), which is defined as $f : \{0,1\}^* \times key \rightarrow GF(2^p)$.
- $\phi_{key}(\cdot)$ – pseudorandom permutation (PRP), which is defined as $\phi : \{0,1\}^{\log_2(l)} \times key \rightarrow \{0,1\}^{\log_2(l)}$.
- ver – a version number bound with the index for individual blocks, which records the times the block has been modified. Initially we assume ver is 0 for all data blocks.
- s_{ij}^{ver} – the seed for PRF, which depends on the file name, block index i, the server position j as well as the optional block version number ver.

III. Ensuring Cloud Data Storage

In cloud data storage system, users store their data in the cloud and no longer possess the data locally. Thus, the correctness and availability of the data files being stored on the distributed cloud servers must be guaranteed. One of the key issues is to effectively detect any unauthorized data modification and corruption, possibly due to server compromise and/or random Byzantine failures. Besides, in the distributed case when such inconsistencies are successfully detected, to find which server the data error lies in is also of great significance, since it can be the first step to fast recover the storage errors.

To address these problems, our main scheme for ensuring cloud data storage is presented in this section. The first part of the section is devoted to a review of basic tools from coding theory that are needed in our scheme for file distribution across cloud servers. Then, the homomorphic token is introduced. The token computation function we are considering belongs to a family of universal hash function [11], chosen to preserve the homomorphic properties, which can be perfectly integrated with the verification of erasure-coded data [8] [12]. Subsequently, it is also shown how to derive a challenge-response protocol for verifying the storage correctness as well as identifying misbehaving servers. Finally, the procedure for file retrieval and error recovery based on erasure-correcting code is outlined.

A. File Distribution Preparation

It is well known that erasure-correcting code may be used to tolerate multiple failures in distributed storage systems. In cloud data storage, we rely on this technique to disperse the data file $\mathbf{F}$ redundantly across a set of $n = m + k$ distributed servers. A $(m + k, k)$ Reed-Solomon erasure-correcting code is used to create k redundancy parity vectors from m data vectors in such a way that the original m data vectors can be reconstructed from any m out of the $m + k$ data and parity vectors. By placing each of the $m + k$ vectors on a different server, the original data file can survive the failure of any k of the $m+k$ servers without any data loss, with a space overhead of k/m. For support of efficient sequential I/O to the original file, our file layout is systematic, i.e., the unmodified m data file vectors together with k parity vectors is distributed across $m + k$ different servers.

Let $\mathbf{F} = (F_1, F_2, \ldots, F_m)$ and $F_i = (f_{1i}, f_{2i}, \ldots, f_{li})^T$ $(i \in \{1, \ldots, m\})$, where $l \leq 2^p - 1$. Note all these blocks are elements of $GF(2^p)$. The systematic layout with parity vectors is achieved with the information dispersal matrix $\mathbf{A}$, derived from an $m \times (m + k)$ Vandermonde matrix [13]:

$$\begin{pmatrix} 1 & 1 & \ldots & 1 & 1 & \ldots & 1 \\ \beta_1 & \beta_2 & \ldots & \beta_m & \beta_{m+1} & \ldots & \beta_n \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \beta_1^{m-1} & \beta_2^{m-1} & \ldots & \beta_m^{m-1} & \beta_{m+1}^{m-1} & \ldots & \beta_n^{m-1} \end{pmatrix},$$

where β_j $(j \in \{1, \ldots, n\})$ are distinct elements randomly picked from $GF(2^p)$.

After a sequence of elementary row transformations, the desired matrix $\mathbf{A}$ can be written as

$$\mathbf{A} = (\mathbf{I}|\mathbf{P}) = \begin{pmatrix} 1 & 0 & \ldots & 0 & p_{11} & p_{12} & \ldots & p_{1k} \\ 0 & 1 & \ldots & 0 & p_{21} & p_{22} & \ldots & p_{2k} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 & p_{m1} & p_{m2} & \ldots & p_{mk} \end{pmatrix},$$

where $\mathbf{I}$ is a $m \times m$ identity matrix and $\mathbf{P}$ is the secret parity generation matrix with size $m \times k$. Note that $\mathbf{A}$ is derived from a Vandermonde matrix, thus it has the property that any m out of the $m + k$ columns form an invertible matrix.

By multiplying $\mathbf{F}$ by $\mathbf{A}$, the user obtains the encoded file:

$$\begin{aligned} \mathbf{G} = \mathbf{F} \cdot \mathbf{A} &= (G^{(1)}, G^{(2)}, \ldots, G^{(m)}, G^{(m+1)}, \ldots, G^{(n)}) \\ &= (F_1, F_2, \ldots, F_m, G^{(m+1)}, \ldots, G^{(n)}), \end{aligned}$$

where $G^{(j)} = (g_1^{(j)}, g_2^{(j)}, \ldots, g_l^{(j)})^T$ $(j \in \{1, \ldots, n\})$. As noticed, the multiplication reproduces the original data file vectors of $\mathbf{F}$ and the remaining part $(G^{(m+1)}, \ldots, G^{(n)})$ are k parity vectors generated based on $\mathbf{F}$.

المنارة للاستشارات
www.manaraa.com

Algorithm 1 Token Pre-computation

```
1: procedure
2:     Choose parameters l, n and function f, φ;
3:     Choose the number t of tokens;
4:     Choose the number r of indices per verification;
5:     Generate master key K_prp and challenge k_chal;
6:     for vector G^(j), j ← 1, n do
7:         for round i ← 1, t do
8:             Derive α_i = f_{k_chal}(i) and k_prp^(i) from K_PRP.
9:             Compute v_i^(j) = Σ_{q=1}^{r} α_i^q * G^(j)[φ_{k_prp^(i)}(q)]
10:        end for
11:    end for
12:    Store all the v_i s locally.
13: end procedure
```

B. Challenge Token Precomputation

In order to achieve assurance of data storage correctness and data error localization simultaneously, our scheme entirely relies on the pre-computed verification tokens. The main idea is as follows: before file distribution the user pre-computes a certain number of short verification tokens on individual vector $G^{(j)}$ $(j \in \{1, \ldots, n\})$, each token covering a random subset of data blocks. Later, when the user wants to make sure the storage correctness for the data in the cloud, he challenges the cloud servers with a set of randomly generated block indices. Upon receiving challenge, each cloud server computes a short "signature" over the specified blocks and returns them to the user. The values of these signatures should match the corresponding tokens pre-computed by the user. Meanwhile, as all servers operate over the same subset of the indices, the requested response values for integrity check must also be a valid codeword determined by secret matrix $\mathbf{P}$.

Suppose the user wants to challenge the cloud servers t times to ensure the correctness of data storage. Then, he must pre-compute t verification tokens for each $G^{(j)}$ $(j \in \{1, \ldots, n\})$, using a PRF $f(\cdot)$, a PRP $\phi(\cdot)$, a challenge key k_{chal} and a master permutation key K_{PRP}. To generate the ith token for server j, the user acts as follows:

1) Derive a random challenge value α_i of $GF(2^p)$ by $\alpha_i = f_{k_{chal}}(i)$ and a permutation key $k_{prp}^{(i)}$ based on K_{PRP}.
2) Compute the set of r randomly-chosen indices:
$$\{I_q \in [1, ..., l] | 1 \le q \le r\}, \text{where } I_q = \phi_{k_{prp}^{(i)}}(q).$$
3) Calculate the token as:
$$v_i^{(j)} = \sum_{q=1}^{r} \alpha_i^q * G^{(j)}[I_q], \text{where } G^{(j)}[I_q] = g_{I_q}^{(j)}.$$

Note that $v_i^{(j)}$, which is an element of $GF(2^p)$ with small size, is the response the user expects to receive from server j when he challenges it on the specified data blocks.

After token generation, the user has the choice of either keeping the pre-computed tokens locally or storing them in encrypted form on the cloud servers. In our case here, the user stores them locally to obviate the need for encryption and lower the bandwidth overhead during dynamic data operation which will be discussed shortly. The details of token generation are shown in Algorithm 1.

Algorithm 2 Correctness Verification and Error Localization

```
1: procedure CHALLENGE(i)
2:     Recompute α_i = f_{k_chal}(i) and k_prp^(i) from K_PRP;
3:     Send {α_i, k_prp^(i)} to all the cloud servers;
4:     Receive from servers:
       {R_i^(j) = Σ_{q=1}^{r} α_i^q * G^(j)[φ_{k_prp^(i)}(q)] | 1 ≤ j ≤ n}
5:     for (j ← m + 1, n) do
6:         R^(j) ← R^(j) − Σ_{q=1}^{r} f_{k_j}(s_{I_q,j}) · α_i^q, I_q = φ_{k_prp^(i)}(q)
7:     end for
8:     if ((R_i^(1), ..., R_i^(m)) · P == (R_i^(m+1), ..., R_i^(n))) then
9:         Accept and ready for the next challenge.
10:    else
11:        for (j ← 1, n) do
12:            if (R_i^(j) != v_i^(j)) then
13:                return server j is misbehaving.
14:            end if
15:        end for
16:    end if
17: end procedure
```

Once all tokens are computed, the final step before file distribution is to blind each parity block $g_i^{(j)}$ in $(G^{(m+1)}, \ldots, G^{(n)})$ by

$$g_i^{(j)} \leftarrow g_i^{(j)} + f_{k_j}(s_{ij}), i \in \{1, \ldots, l\},$$

where k_j is the secret key for parity vector $G^{(j)}$ $(j \in \{m+1, \ldots, n\})$. This is for protection of the secret matrix $\mathbf{P}$. We will discuss the necessity of using blinded parities in detail in Section V. After blinding the parity information, the user disperses all the n encoded vectors $G^{(j)}$ $(j \in \{1, \ldots, n\})$ across the cloud servers $S_1, S_2, \ldots, S_n$.

C. Correctness Verification and Error Localization

Error localization is a key prerequisite for eliminating errors in storage systems. However, many previous schemes do not explicitly consider the problem of data error localization, thus only provide binary results for the storage verification. Our scheme outperforms those by integrating the correctness verification and error localization in our challenge-response protocol: the response values from servers for each challenge not only determine the correctness of the distributed storage, but also contain information to locate potential data error(s).

Specifically, the procedure of the i-th challenge-response for a cross-check over the n servers is described as follows:

1) The user reveals the α_i as well as the i-th permutation key $k_{prp}^{(i)}$ to each servers.
2) The server storing vector $G^{(j)}$ aggregates those r rows specified by index $k_{prp}^{(i)}$ into a linear combination
$$R_i^{(j)} = \sum_{q=1}^{r} \alpha_i^q * G^{(j)}[\phi_{k_{prp}^{(i)}}(q)].$$

المنارة للاستشارات
www.manaraa.com

3) Upon receiving $R_i^{(j)}$s from all the servers, the user takes away blind values in $R^{(j)}$ $(j \in \{m+1, \ldots, n\})$ by

$$R_i^{(j)} \leftarrow R_i^{(j)} - \sum_{q=1}^{r} f_{k_j}(s_{I_q,j}) \cdot \alpha_i^q, \text{where } I_q = \phi_{k_{prp}^{(i)}}(q).$$

4) Then the user verifies whether the received values remain a valid codeword determined by secret matrix $\mathbf{P}$:

$$(R_i^{(1)}, \ldots, R_i^{(m)}) \cdot \mathbf{P} \stackrel{?}{=} (R_i^{(m+1)}, \ldots, R_i^{(n)}).$$

Because all the servers operate over the same subset of indices, the linear aggregation of these r specified rows $(R_i^{(1)}, \ldots, R_i^{(n)})$ has to be a codeword in the encoded file matrix. If the above equation holds, the challenge is passed. Otherwise, it indicates that among those specified rows, there exist file block corruptions.

Once the inconsistency among the storage has been successfully detected, we can rely on the pre-computed verification tokens to further determine where the potential data error(s) lies in. Note that each response $R_i^{(j)}$ is computed exactly in the same way as token $v_i^{(j)}$, thus the user can simply find which server is misbehaving by verifying the following n equations:

$$R_i^{(j)} \stackrel{?}{=} v_i^{(j)}, j \in \{1, \ldots, n\}.$$

Algorithm 2 gives the details of correctness verification and error localization.

D. File Retrieval and Error Recovery

Since our layout of file matrix is systematic, the user can reconstruct the original file by downloading the data vectors from the first m servers, assuming that they return the correct response values. Notice that our verification scheme is based on random spot-checking, so the storage correctness assurance is a probabilistic one. However, by choosing system parameters $(e.g., r, l, t)$ appropriately and conducting enough times of verification, we can guarantee the successful file retrieval with high probability. On the other hand, whenever the data corruption is detected, the comparison of pre-computed tokens and received response values can guarantee the identification of misbehaving server(s), again with high probability, which will be discussed shortly. Therefore, the user can always ask servers to send back blocks of the r rows specified in the challenge and regenerate the correct blocks by erasure correction, shown in Algorithm 3, as long as there are at most k misbehaving servers are identified. The newly recovered blocks can then be redistributed to the misbehaving servers to maintain the correctness of storage.

Algorithm 3 Error Recovery

1: **procedure**
   % Assume the block corruptions have been detected among
   % the specified r rows;
   % Assume $s \leq k$ servers have been identified misbehaving
2: Download r rows of blocks from servers;
3: Treat s servers as erasures and recover the blocks.
4: Resend the recovered blocks to corresponding servers.
5: **end procedure**

IV. Providing Dynamic Data Operation Support

So far, we assumed that $\mathbf{F}$ represents static or archived data. This model may fit some application scenarios, such as libraries and scientific datasets. However, in cloud data storage, there are many potential scenarios where data stored in the cloud is dynamic, like electronic documents, photos, or log files etc. Therefore, it is crucial to consider the dynamic case, where a user may wish to perform various block-level operations of update, delete and append to modify the data file while maintaining the storage correctness assurance.

The straightforward and trivial way to support these operations is for user to download all the data from the cloud servers and re-compute the whole parity blocks as well as verification tokens. This would clearly be highly inefficient. In this section, we will show how our scheme can explicitly and efficiently handle dynamic data operations for cloud data storage.

A. Update Operation

In cloud data storage, sometimes the user may need to modify some data block(s) stored in the cloud, from its current value f_{ij} to a new one, $f_{ij} + \Delta f_{ij}$. We refer this operation as data update. Due to the linear property of Reed-Solomon code, a user can perform the update operation and generate the updated parity blocks by using Δf_{ij} only, without involving any other unchanged blocks. Specifically, the user can construct a general update matrix $\Delta\mathbf{F}$ as

$$\begin{aligned} \Delta\mathbf{F} &= \begin{pmatrix} \Delta f_{11} & \Delta f_{12} & \ldots & \Delta f_{1m} \\ \Delta f_{21} & \Delta f_{22} & \ldots & \Delta f_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ \Delta f_{l1} & \Delta f_{l2} & \ldots & \Delta f_{lm} \end{pmatrix} \\ &= (\Delta F_1, \Delta F_2, \ldots, \Delta F_m). \end{aligned}$$

Note that we use zero elements in $\Delta\mathbf{F}$ to denote the unchanged blocks. To maintain the corresponding parity vectors as well as be consistent with the original file layout, the user can multiply $\Delta\mathbf{F}$ by $\mathbf{A}$ and thus generate the update information for both the data vectors and parity vectors as follows:

$$\begin{aligned} \Delta\mathbf{F} \cdot \mathbf{A} &= (\Delta G^{(1)}, \ldots, \Delta G^{(m)}, \Delta G^{(m+1)}, \ldots, \Delta G^{(n)}) \\ &= (\Delta F_1, \ldots, \Delta F_m, \Delta G^{(m+1)}, \ldots, \Delta G^{(n)}), \end{aligned}$$

where $\Delta G^{(j)}$ $(j \in \{m+1, \ldots, n\})$ denotes the update information for the parity vector $G^{(j)}$.

Because the data update operation inevitably affects some or all of the remaining verification tokens, after preparation of update information, the user has to amend those unused tokens for each vector $G^{(j)}$ to maintain the same storage correctness assurance. In other words, for all the unused tokens, the user needs to exclude every occurrence of the old data block and replace it with the new one. Thanks to the homomorphic construction of our verification token, the user can perform the token update efficiently. To give more details, suppose a

المنارة للاستشارات
www.manaraa.com

block $G^{(j)}[I_s]$, which is covered by the specific token $v_i^{(j)}$, has been changed to $G^{(j)}[I_s] + \Delta G^{(j)}[I_s]$, where $I_s = \phi_{k_{prp}^{(i)}}(s)$. To maintain the usability of token $v_i^{(j)}$, it is not hard to verify that the user can simply update it by

$$v_i^{(j)} \leftarrow v_i^{(j)} + \alpha_i^s * \Delta G^{(j)}[I_s],$$

without retrieving any other $r-1$ blocks required in the pre-computation of $v_i^{(j)}$.

After the amendment to the affected tokens[1], the user needs to blind the update information $\Delta g_i^{(j)}$ for each parity block in $(\Delta G^{(m+1)}, \ldots, \Delta G^{(n)})$ to hide the secret matrix $\mathbf{P}$ by

$$\Delta g_i^{(j)} \leftarrow \Delta g_i^{(j)} + f_{k_j}(s_{ij}^{ver}), i \in \{1, \ldots, l\}.$$

Here we use a new seed s_{ij}^{ver} for the PRF. The version number ver functions like a counter which helps the user to keep track of the blind information on the specific parity blocks. After blinding, the user sends update information to the cloud servers, which perform the update operation as

$$G^{(j)} \leftarrow G^{(j)} + \Delta G^{(j)}, \ (j \in \{1, \ldots, n\}).$$

B. Delete Operation

Sometimes, after being stored in the cloud, certain data blocks may need to be deleted. The delete operation we are considering is a general one, in which user replaces the data block with zero or some special reserved data symbol. From this point of view, the delete operation is actually a special case of the data update operation, where the original data blocks can be replaced with zeros or some predetermined special blocks. Therefore, we can rely on the update procedure to support delete operation, i.e., by setting Δf_{ij} in ΔF to be $-\Delta f_{ij}$. Also, all the affected tokens have to be modified and the updated parity information has to be blinded using the same method specified in update operation.

C. Append Operation

In some cases, the user may want to increase the size of his stored data by adding blocks at the end of the data file, which we refer as data append. We anticipate that the most frequent append operation in cloud data storage is bulk append, in which the user needs to upload a large number of blocks (not a single block) at one time.

Given the file matrix $\mathbf{F}$ illustrated in file distribution preparation, appending blocks towards the end of a data file is equivalent to concatenate corresponding rows at the bottom of the matrix layout for file $\mathbf{F}$. In the beginning, there are only l rows in the file matrix. To simplify the presentation, we suppose the user wants to append m blocks at the end of file $\mathbf{F}$, denoted as $(f_{l+1,1}, f_{l+1,2}, ..., f_{l+1,m})$ (We can always use zero-padding to make a row of m elements.). With the secret matrix $\mathbf{P}$, the user can directly calculate the append blocks for each parity server as

$$(f_{l+1,1}, ..., f_{l+1,m}) \cdot \mathbf{P} = (g_{l+1}^{(m+1)}, ..., g_{l+1}^{(n)}).$$

[1] In practice, it is possible that only a fraction of tokens need amendment, since the updated blocks may not be covered by all the tokens.

To support block append operation, we need a slight modification to our token pre-computation. Specifically, we require the user to expect the maximum size in blocks, denoted as l_{max}, for each of his data vector. The idea of supporting block append, which is similar as adopted in [7], relies on the initial budget for the maximum anticipated data size l_{max} in each encoded data vector as well as the system parameter $r_{max} = \lceil r * (l_{max}/l) \rceil$ for each pre-computed challenge-response token. The pre-computation of the i-th token on server j is modified as follows:

$$v_i = \sum_{q=1}^{r_{max}} \alpha_i^q * G^{(j)}[I_q],$$

where

$$G^{(j)}[I_q] = \begin{cases} G^{(j)}[\phi_{k_{prp}^{(i)}}(q)] & , [\phi_{k_{prp}^{(i)}}(q)] \leq l \\ 0 & , [\phi_{k_{prp}^{(i)}}(q)] > l \end{cases}$$

This formula guarantees that on average, there will be r indices falling into the range of existing l blocks. Because the cloud servers and the user have the agreement on the number of existing blocks in each vector $G^{(j)}$, servers will follow exactly the above procedure when re-computing the token values upon receiving user's challenge request.

Now when the user is ready to append new blocks, i.e., both the file blocks and the corresponding parity blocks are generated, the total length of each vector $G^{(j)}$ will be increased and fall into the range $[l, l_{max}]$. Therefore, the user will update those affected tokens by adding $\alpha_i^s * G^{(j)}[I_s]$ to the old v_i whenever $G^{(j)}[I_s] \neq 0$ for $I_s > l$, where $I_s = \phi_{k_{prp}^{(i)}}(s)$. The parity blinding is similar as introduced in update operation, thus is omitted here.

D. Insert Operation

An insert operation to the data file refers to an append operation at the desired index position while maintaining the same data block structure for the whole data file, i.e., inserting a block $F[j]$ corresponds to shifting all blocks starting with index $j+1$ by one slot. An insert operation may affect many rows in the logical data file matrix $\mathbf{F}$, and a substantial number of computations are required to renumber all the subsequent blocks as well as re-compute the challenge-response tokens. Therefore, an efficient insert operation is difficult to support and thus we leave it for our future work.

V. Security Analysis and Performance Evaluation

In this section, we analyze our proposed scheme in terms of security and efficiency. Our security analysis focuses on the adversary model defined in Section II. We also evaluate the efficiency of our scheme via implementation of both file distribution preparation and verification token precomputation.

A. Security Strength Against Weak Adversary

1) Detection Probability against data modification: In our scheme, servers are required to operate on specified rows in each correctness verification for the calculation of requested

للاستشارات المنارة
www.manaraa.com

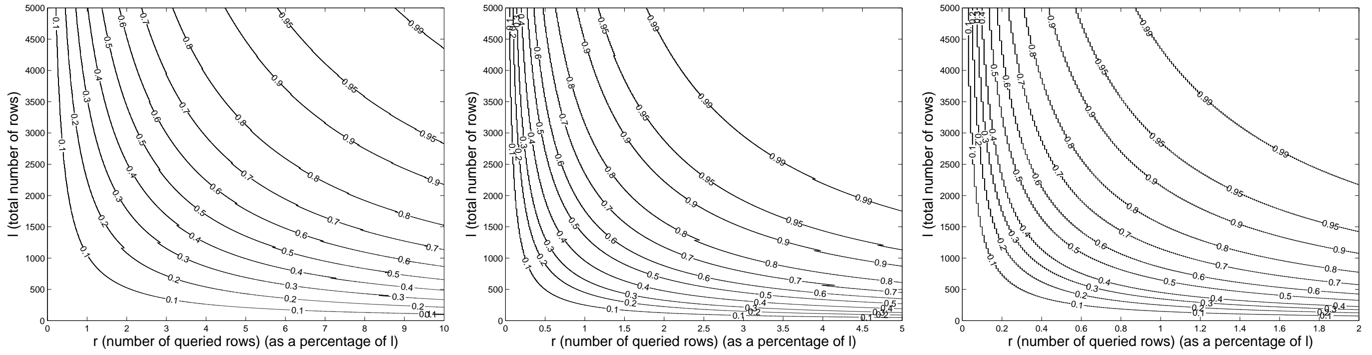

Fig. 2: The detection probability P_d against data modification. We show P_d as a function of l (the number of blocks on each cloud storage server) and r (the number of rows queried by the user, shown as a percentage of l) for three values of z (the number of rows modified by the adversary). left) $z = 1\%$ of l; middle) $z = 5\%$ of l; right) $z = 10\%$ of l. Note that all graphs are plotted under $p = 8$, $n_c = 10$ and $k = 5$ and each graph has a different scale.

token. We will show that this "sampling" strategy on selected rows instead of all can greatly reduce the computational overhead on the server, while maintaining the detection of the data corruption with high probability.

Suppose n_c servers are misbehaving due to the possible compromise or Byzantine failure. In the following analysis, we do not limit the value of n_c, i.e., $n_c \leq n$. Assume the adversary modifies the data blocks in z rows out of the l rows in the encoded file matrix. Let r be the number of different rows for which the user asks for check in a challenge. Let X be a discrete random variable that is defined to be the number of rows chosen by the user that matches the rows modified by the adversary. We first analyze the matching probability that at least one of the rows picked by the user matches one of the rows modified by the adversary:

$$\begin{aligned} P_m^r &= 1 - P\{X = 0\} \\ &= 1 - \prod_{i=0}^{r-1}(1 - \min\{\frac{z}{l-i}, 1\}) \\ &\geq 1 - (\frac{l-z}{l})^r. \end{aligned}$$

Note that if none of the specified r rows in the i-th verification process are deleted or modified, the adversary avoids the detection.

Next, we study the probability of a false negative result that the data blocks in those specified r rows has been modified, but the checking equation still holds. Consider the responses $R_i^{(1)}, \ldots, R_i^{(n)}$ returned from the data storage servers for the i-th challenge, each response value $R_i^{(j)}$, calculated within $GF(2^p)$, is based on r blocks on server j. The number of responses $R^{(m+1)}, \ldots, R^{(n)}$ from parity servers is $k = n - m$. Thus, according to the proposition 2 of our previous work in [14], the false negative probability is

$$P_f^r = Pr_1 + Pr_2,$$

where $Pr_1 = \frac{(1+2^{-p})^{n_c} - 1}{2^{n_c} - 1}$ and $Pr_2 = (1 - \mathrm{Pr}_1)(2^{-p})^k$.

Based on above discussion, it follows that the probability of data modification detection across all storage servers is

$$P_d = P_m^r \cdot (1 - P_f^r).$$

Figure 2 plots P_d for different values of l, r, z while we set $p = 8, n_c = 10$ and $k = 5$. [2] From the figure we can see that if more than a fraction of the data file is corrupted, then it suffices to challenge for a small constant number of rows in order to achieve detection with high probability. For example, if $z = 1\%$ of l, every token only needs to cover 460 indices in order to achieve the detection probability of at least 99%.

2) Identification Probability for Misbehaving Servers: We have shown that, if the adversary modifies the data blocks among any of the data storage servers, our sampling checking scheme can successfully detect the attack with high probability. As long as the data modification is caught, the user will further determine which server is malfunctioning. This can be achieved by comparing the response values $R_i^{(j)}$ with the pre-stored tokens $v_i^{(j)}$, where $j \in \{1, \ldots, n\}$. The probability for error localization or identifying misbehaving server(s) can be computed in a similar way. It is the product of the matching probability for sampling check and the probability of complementary event for the false negative result. Obviously, the matching probability is $\widehat{P}_m^r = 1 - \prod_{i=0}^{r-1}(1 - \min\{\frac{\hat{z}}{l-i}, 1\})$, where $\hat{z} \leq z$.

Next, we consider the false negative probability that $R_i^{(j)} = v_i^{(j)}$ when at least one of $\hat{z}$ blocks is modified. According to proposition 1 of [14], tokens calculated in $GF(2^p)$ for two different data vectors collide with probability $\widehat{P}_f^r = 2^{-p}$. Thus, the identification probability for misbehaving server(s) is

$$\widehat{P}_d = \widehat{P}_m^r \cdot (1 - \widehat{P}_f^r).$$

Along with the analysis in detection probability, if $z = 1\%$ of l and each token covers 460 indices, the identification probability for misbehaving servers is at least 99%.

[2]Note that n_c and k only affect the false negative probability P_f^r. However in our scheme, since $p = 8$ almost dominates the negligibility of P_f^r, the value of n_c and k have little effect in the plot of P_d.

المنارة للاستشارات
www.manaraa.com

B. Security Strength Against Strong Adversary

In this section, we analyze the security strength of our schemes against server colluding attack and explain why blinding the parity blocks can help improve the security strength of our proposed scheme.

Recall that in the file distribution preparation, the redundancy parity vectors are calculated via multiplying the file matrix $\mathbf{F}$ by $\mathbf{P}$, where $\mathbf{P}$ is the secret parity generation matrix we later rely on for storage correctness assurance. If we disperse all the generated vectors directly after token precomputation, i.e., without blinding, malicious servers that collaborate can reconstruct the secret $\mathbf{P}$ matrix easily: they can pick blocks from the same rows among the data and parity vectors to establish a set of $m \cdot k$ linear equations and solve for the $m \cdot k$ entries of the parity generation matrix $\mathbf{P}$. Once they have the knowledge of $\mathbf{P}$, those malicious servers can consequently modify any part of the data blocks and calculate the corresponding parity blocks, and vice versa, making their codeword relationship always consistent. Therefore, our storage correctness challenge scheme would be undermined—even if those modified blocks are covered by the specified rows, the storage correctness check equation would always hold.

To prevent colluding servers from recovering $\mathbf{P}$ and making up consistently-related data and parity blocks, we utilize the technique of adding random perturbations to the encoded file matrix and hence hide the secret matrix $\mathbf{P}$. We make use of a keyed pseudorandom function $f_{k_j}(\cdot)$ with key k_j and seed s_{ij}^{ver}, both of which has been introduced previously. In order to maintain the systematic layout of data file, we only blind the parity blocks with random perturbations. Our purpose is to add "noise" to the set of linear equations and make it computationally infeasible to solve for the correct secret matrix $\mathbf{P}$. By blinding each parity block with random perturbation, the malicious servers no longer have all the necessary information to build up the correct linear equation groups and therefore cannot derive the secret matrix $\mathbf{P}$.

C. Performance Evaluation

1) File Distribution Preparation: We implemented the generation of parity vectors for our scheme under field $GF(2^8)$. Our experiment is conducted using C on a system with an Intel Core 2 processor running at 1.86 GHz, 2048 MB of RAM, and a 7200 RPM Western Digital 250 GB Serial ATA drive with an 8 MB buffer. We consider two sets of different parameters for the $(m+k, m)$ Reed-Solomon encoding. Table I shows the average encoding cost over 10 trials for an 8 GB file. In the table on the top, we set the number of parity vectors constant at 2. In the one at the bottom, we keep the number of the data vectors fixed at 8, and increase the number of parity vectors. Note that as m increases, the length l of data vectors on each server will decrease, which results in fewer calls to the Reed-Solomon encoder. Thus the cost in the top table decreases when more data vectors are involved. From Table I, it can be shown that the performance of our scheme is comparable to that of [10], even if our scheme supports dynamic data operation while [10] is for static data only.

set I	$m=4$	$m=6$	$m=8$	$m=10$
$k=2$	567.45s	484.55s	437.22s	414.22s

set II	$k=1$	$k=2$	$k=3$	$k=4$
$m=8$	358.90s	437.22s	584.55s	733.34s

TABLE I: The cost of parity generation in seconds for an 8GB data file. For set I, the number of parity servers k is fixed; for set II, the number of data servers m is constant.

2) Challenge Token Pre-computation: Although in our scheme the number of verification token t is a fixed priori determined before file distribution, we can overcome this issue by choosing sufficient large t in practice. For example, when t is selected to be 1825 and 3650, the data file can be verified every day for the next 5 years and 10 years, respectively. Following the security analysis, we select a practical parameter $r = 460$ for our token pre-computation (see previous subsections), i.e., each token covers 460 different indices. Other parameters are along with the file distribution preparation. According to our implementation, the average token pre-computation cost for $t = 1825$ is 51.97s per data vector, and for $t = 3650$ is 103.94s per data vector. This is faster than the hash function based token pre-computation scheme proposed in [7]. For a typical number of 8 servers, the total cost for token pre-computation is no more than 15 minutes. Note that each token is only an element of field $GF(2^8)$, the extra storage for those pre-computed tokens is less than 1MB, and thus can be neglected.

VI. Related Work

Juels et al. [3] described a formal "proof of retrievability" (POR) model for ensuring the remote data integrity. Their scheme combines spot-cheking and error-correcting code to ensure both possession and retrievability of files on archive service systems. Shacham et al. [4] built on this model and constructed a random linear function based homomorphic authenticator which enables unlimited number of queries and requires less communication overhead. Bowers et al. [5] proposed an improved framework for POR protocols that generalizes both Juels and Shacham's work. Later in their subsequent work, Bowers et al. [10] extended POR model to distributed systems. However, all these schemes are focusing on static data. The effectiveness of their schemes rests primarily on the preprocessing steps that the user conducts before outsourcing the data file F. Any change to the contents of F, even few bits, must propagate through the error-correcting code, thus introducing significant computation and communication complexity.

Ateniese et al. [6] defined the "provable data possession" (PDP) model for ensuring possession of file on untrusted storages. Their scheme utilized public key based homomorphic tags for auditing the data file, thus providing public verifiability. However, their scheme requires sufficient computation overhead that can be expensive for an entire file. In their subsequent work, Ateniese et al. [7] described a PDP scheme

المنارة للاستشارات
www.manaraa.com

that uses only symmetric key cryptography. This method has lower-overhead than their previous scheme and allows for block updates, deletions and appends to the stored file, which has also been supported in our work. However, their scheme focuses on single server scenario and does not address small data corruptions, leaving both the distributed scenario and data error recovery issue unexplored. Curtmola et al. [15] aimed to ensure data possession of multiple replicas across the distributed storage system. They extended the PDP scheme to cover multiple replicas without encoding each replica separately, providing guarantee that multiple copies of data are actually maintained.

In other related work, Lillibridge et al. [9] presented a P2P backup scheme in which blocks of a data file are dispersed across $m+k$ peers using an $(m+k, m)$-erasure code. Peers can request random blocks from their backup peers and verify the integrity using separate keyed cryptographic hashes attached on each block. Their scheme can detect data loss from free-riding peers, but does not ensure all data is unchanged. Filho et al. [16] proposed to verify data integrity using RSA-based hash to demonstrate uncheatable data possession in peer-to-peer file sharing networks. However, their proposal requires exponentiation over the entire data file, which is clearly impractical for the server whenever the file is large. Shah et al. [17] proposed allowing a TPA to keep online storage honest by first encrypting the data then sending a number of pre-computed symmetric-keyed hashes over the encrypted data to the auditor. However, their scheme only works for encrypted files, and auditors must maintain long-term state. Schwarz et al. [8] proposed to ensure file integrity across multiple distributed servers, using erasure-coding and block-level file integrity checks. However, their scheme only considers static data files and do not explicitly study the problem of data error localization, which we are considering in this work.

VII. Conclusion

In this paper, we investigated the problem of data security in cloud data storage, which is essentially a distributed storage system. To ensure the correctness of users' data in cloud data storage, we proposed an effective and flexible distributed scheme with explicit dynamic data support, including block update, delete, and append. We rely on erasure-correcting code in the file distribution preparation to provide redundancy parity vectors and guarantee the data dependability. By utilizing the homomorphic token with distributed verification of erasure-coded data, our scheme achieves the integration of storage correctness insurance and data error localization, i.e., whenever data corruption has been detected during the storage correctness verification across the distributed servers, we can almost guarantee the simultaneous identification of the misbehaving server(s). Through detailed security and performance analysis, we show that our scheme is highly efficient and resilient to Byzantine failure, malicious data modification attack, and even server colluding attacks.

We believe that data storage security in Cloud Computing, an area full of challenges and of paramount importance, is still in its infancy now, and many research problems are yet to be identified. We envision several possible directions for future research on this area. The most promising one we believe is a model in which public verifiability is enforced. Public verifiability, supported in [6] [4] [17], allows TPA to audit the cloud data storage without demanding users' time, feasibility or resources. An interesting question in this model is if we can construct a scheme to achieve both public verifiability and storage correctness assurance of dynamic data. Besides, along with our research on dynamic cloud data storage, we also plan to investigate the problem of fine-grained data error localization.

Acknowledgement

This work was supported in part by the US National Science Foundation under grant CNS-0831963, CNS-0626601, CNS-0716306, and CNS-0831628.

References

[1] Amazon.com, "Amazon Web Services (AWS)," Online at http://aws.amazon.com, 2008.
[2] N. Gohring, "Amazon's S3 down for several hours," Online at http://www.pcworld.com/businesscenter/article/142549/amazons_s3_down_for_several_hours.html, 2008.
[3] A. Juels and J. Burton S. Kaliski, "PORs: Proofs of Retrievability for Large Files," *Proc. of CCS '07*, pp. 584–597, 2007.
[4] H. Shacham and B. Waters, "Compact Proofs of Retrievability," *Proc. of Asiacrypt '08*, Dec. 2008.
[5] K. D. Bowers, A. Juels, and A. Oprea, "Proofs of Retrievability: Theory and Implementation," Cryptology ePrint Archive, Report 2008/175, 2008, http://eprint.iacr.org/.
[6] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable Data Possession at Untrusted Stores," *Proc. of CCS '07*, pp. 598–609, 2007.
[7] G. Ateniese, R. D. Pietro, L. V. Mancini, and G. Tsudik, "Scalable and Efficient Provable Data Possession," *Proc. of SecureComm '08*, pp. 1–10, 2008.
[8] T. S. J. Schwarz and E. L. Miller, "Store, Forget, and Check: Using Algebraic Signatures to Check Remotely Administered Storage," *Proc. of ICDCS '06*, pp. 12–12, 2006.
[9] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, and M. Isard, "A Cooperative Internet Backup Scheme," *Proc. of the 2003 USENIX Annual Technical Conference (General Track)*, pp. 29–41, 2003.
[10] K. D. Bowers, A. Juels, and A. Oprea, "HAIL: A High-Availability and Integrity Layer for Cloud Storage," Cryptology ePrint Archive, Report 2008/489, 2008, http://eprint.iacr.org/.
[11] L. Carter and M. Wegman, "Universal Hash Functions," *Journal of Computer and System Sciences*, vol. 18, no. 2, pp. 143–154, 1979.
[12] J. Hendricks, G. Ganger, and M. Reiter, "Verifying Distributed Erasure-coded Data," *Proc. 26th ACM Symposium on Principles of Distributed Computing*, pp. 139–146, 2007.
[13] J. S. Plank and Y. Ding, "Note: Correction to the 1997 Tutorial on Reed-Solomon Coding," University of Tennessee, Tech. Rep. CS-03-504, 2003.
[14] Q. Wang, K. Ren, W. Lou, and Y. Zhang, "Dependable and Secure Sensor Data Storage with Dynamic Integrity Assurance," *Proc. of IEEE INFOCOM*, 2009.
[15] R. Curtmola, O. Khan, R. Burns, and G. Ateniese, "MR-PDP: Multiple-Replica Provable Data Possession," *Proc. of ICDCS '08*, pp. 411–420, 2008.
[16] D. L. G. Filho and P. S. L. M. Barreto, "Demonstrating Data Possession and Uncheatable Data Transfer," Cryptology ePrint Archive, Report 2006/150, 2006, http://eprint.iacr.org/.
[17] M. A. Shah, M. Baker, J. C. Mogul, and R. Swaminathan, "Auditing to Keep Online Storage Services Honest," *Proc. 11th USENIX Workshop on Hot Topics in Operating Systems (HOTOS '07)*, pp. 1–6, 2007.

المنارة للاستشارات
www.manaraa.com